**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #3**

Due: September 20, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

In this lab, we implemented different versions of sorting algorithms. We used sorting algorithms such as quicksort, and bubble sort. For this lab, it is important to review sorting algorithms, and using recursive functions versus iterative functions or vice-versa. The main purpose of this lab was to find different ways of implementing sorting algorithms, such as not using recursion or implementing quicksort with stacks, and then finding the kth smallest element in a sorted list.

**Proposed Solution Design and Implementation**

**Part #1:** For the first part, I used code from a previous assignment in which I implemented regular quicksort. I modified this code into a more efficient quicksort algorithm that partitions a list by choosing the last element as a pivot, and a partition is made based off of that pivot. The list is then sorted and the kth element is returned. For the modified quicksort, I used the same partition algorithm, however once sorting, I modified it to only sort the part of the partitioned list that contains the kth element. I did this by comparing the pivot value to the kth element that needed to be found. If it was less than the pivot, then only one half of the list was sorted, and if it was greater than the pivot, then the other half was only sorted. As for the bubble sort, I iterated through the list and if the current element was greater than the next, I swapped the elements.

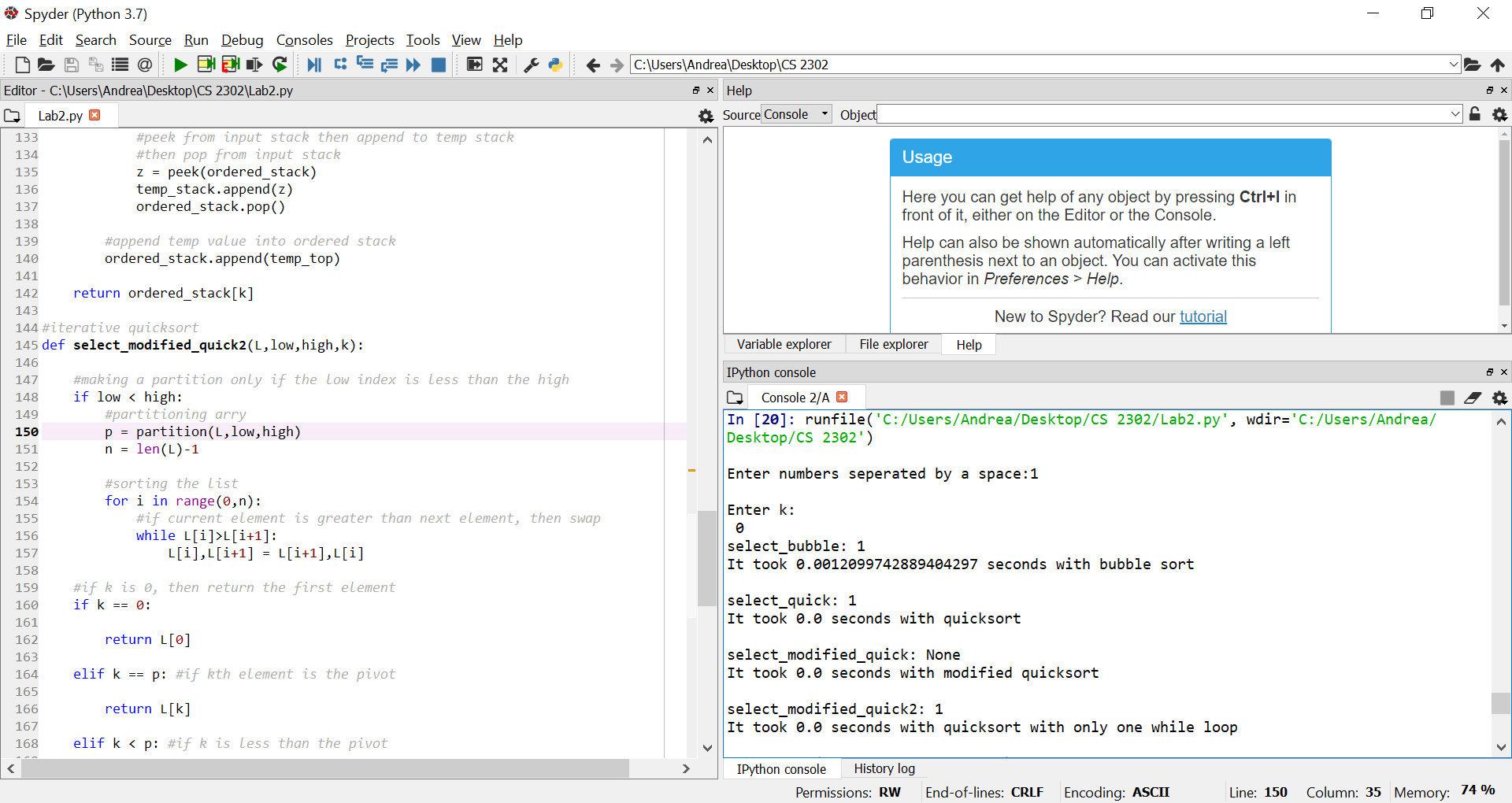
**Part #2:** As for part 2, I also implemented the same partition algorithm, however once I started sorting, I only implemented the sorting algorithm with stacks instead of recursion. To achieve this, I created a few helper functions that checked if the stack was empty and one that gave me the top of the stack. After the partition was done, I sorted by creating two stacks and appending and removing back and forth from them. I first added the entire list of numbers to a stack, then added the numbers to the input stack by checking if they were greater or less than each other. I then found the kth element by simply returning the value at the position of k after sorting. For the quicksort implementation using only a iterative solution, I first partitioned the list then I looped through the entire partition and checked if the current element was greater than the next and swapped them if this was true. I then, again, returned the kth element in a similar way to every other quicksort implementation.

**Experimental Results**

To test the program, I tested each function with a variety of test cases that contained different values for n. I then compared all running times to determine which algorithm executed faster with the different input values of n.

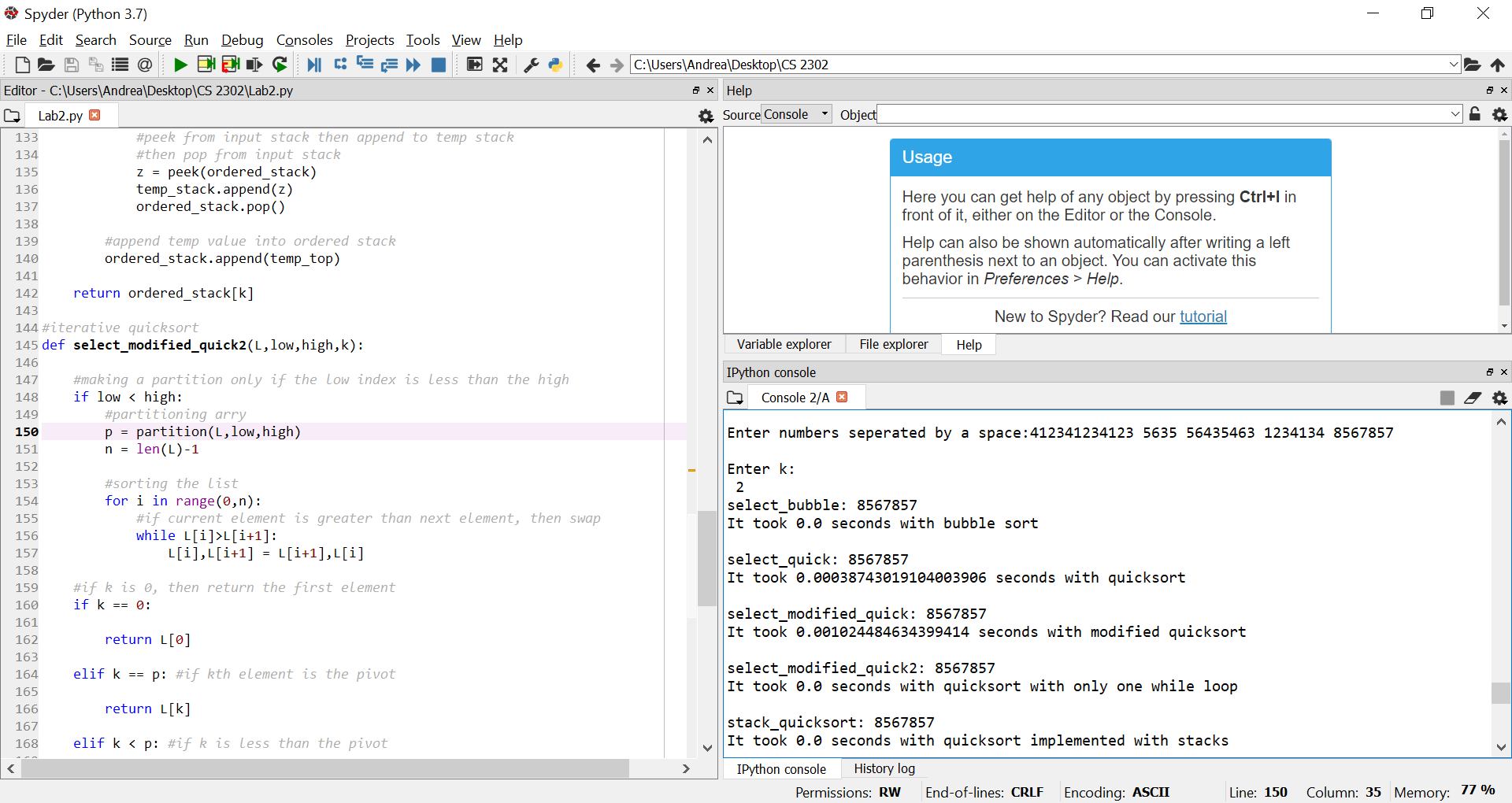
|  |  |
| --- | --- |
| **Sorting Algorithm** | **Big-O running time** |
| **select\_bubble()** | **O(n^2)** |
| **select\_quick()** | **O(log(n))** |
| **select\_modified\_quick()** | **O(n)** |
| **select\_modified\_quick2()** | **O(n)** |
| **stack\_quicksort()** | **O(n)** |

**Test case #1:** The first test case was a list that only contained one element. I entered only the number 1 for this test case, and entered k as the number 0. This determines whether the program will know how to handle only one number and to see if it does any extra steps. It will also determine which algorithm is faster.



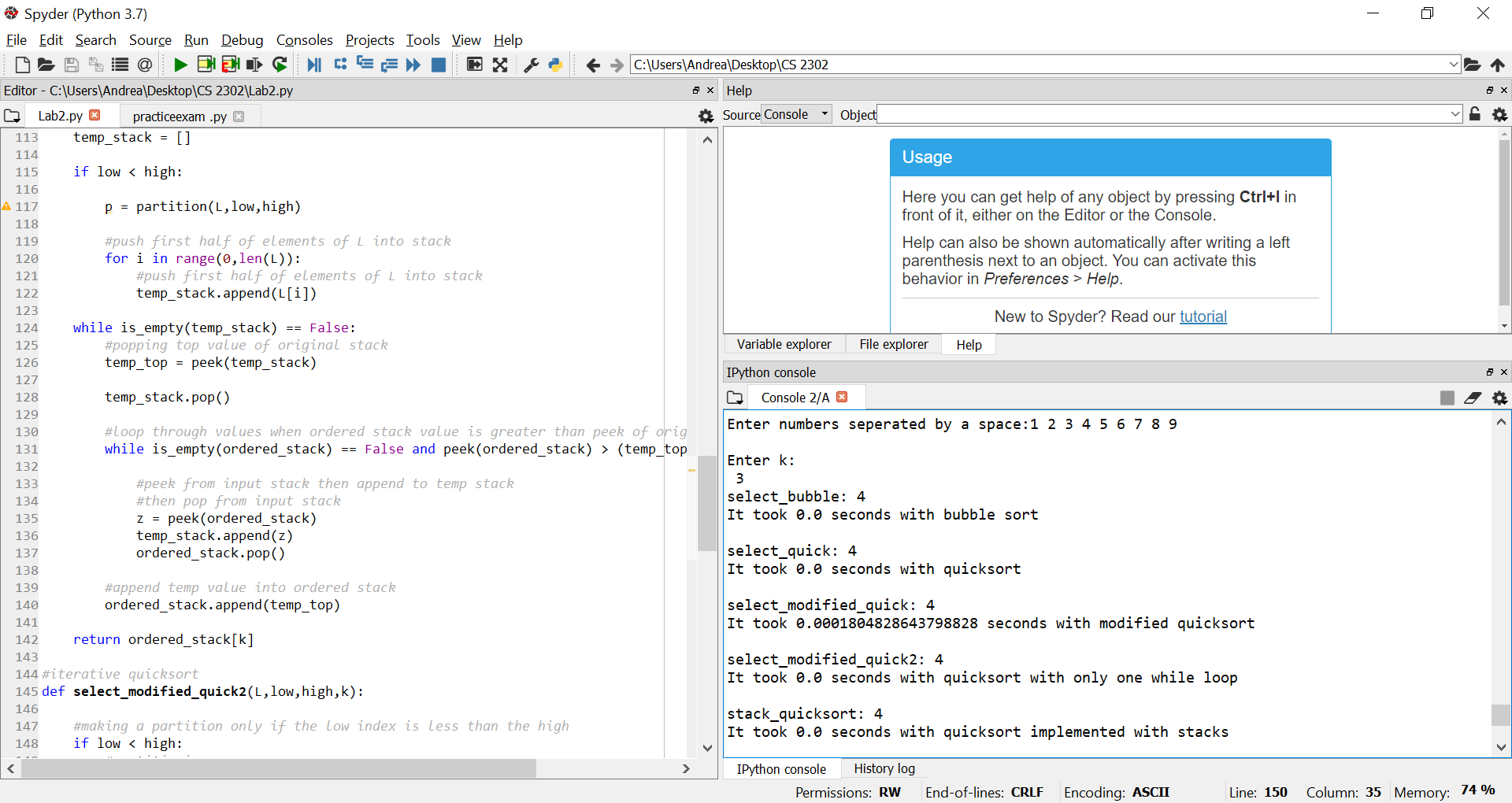
* No error, all methods returned the correct element, and all of them took zero seconds to execute, except for bubble sort which took a little more than zero seconds. This makes sense since bubble sort has Big-O running time of O(n^2).

**Test case #2:** The second test case consisted of a list that contained very large numbers. The value that needed to be returned was the value at index 2.



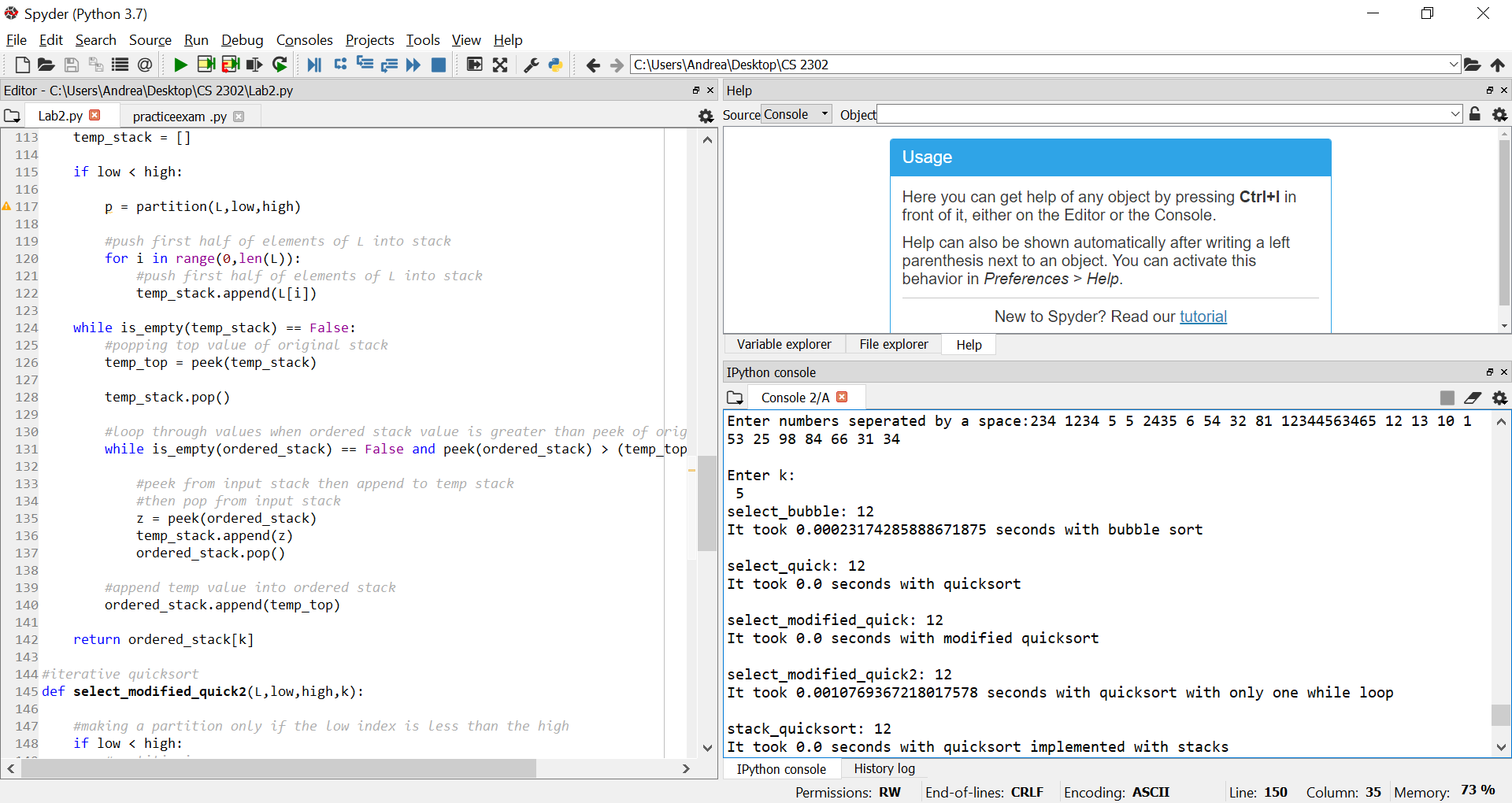
* No error, the program did take a little longer to sort, however it still returned the correct value of k. The fastest algorithm was the quicksort algorithm implemented with a stack.

**Test case #3:** The third test case contained an already sorted list of numbers.



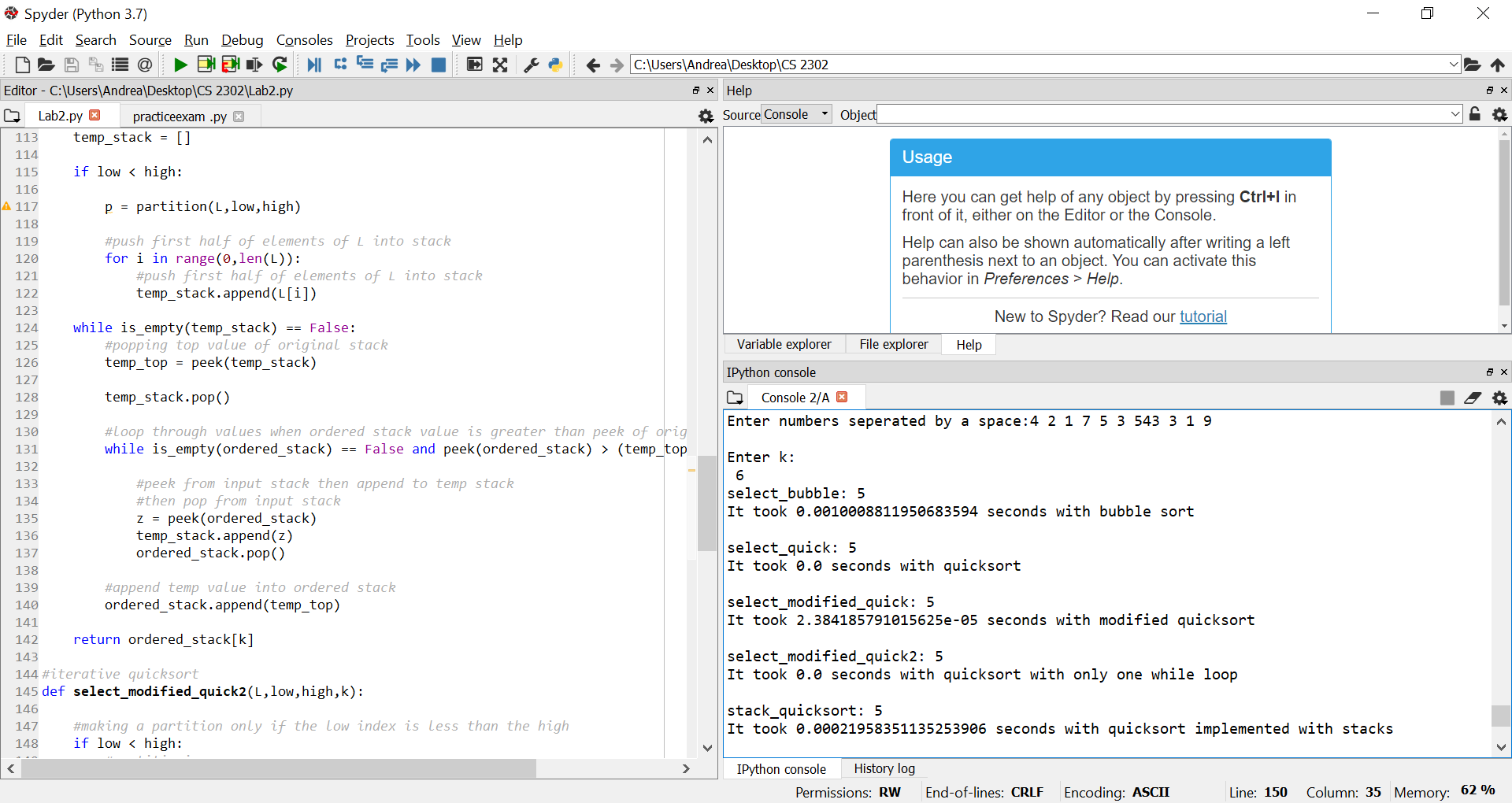
* The slowest algorithm within this test case was the modified quicksort which only has one recursive call. This is due to the fact that the function contains many checks to find where the value at k is, therefore making the execution take longer.

**Test case #4:** The fourth test case contained a list of length 20, this determined how fast the list can handle a large amount of numbers.



* The modified version of quicksort and the quicksort implementation with a stack were the fastest in this scenario. The slowest algorithm was bubble sort. This occurred since bubble sort has a Big-O running time of O(n) which is usually slow.

**Test case #5:** For the final case I used a normal unordered list that contained duplicate numbers. The length of the list is 10.



* The normal quicksort and the quicksort with only a while loop performed the operation the quickest. The slowest running time was 2 seconds which was the quicksort implementation with only one recursive call.

**Running times:**

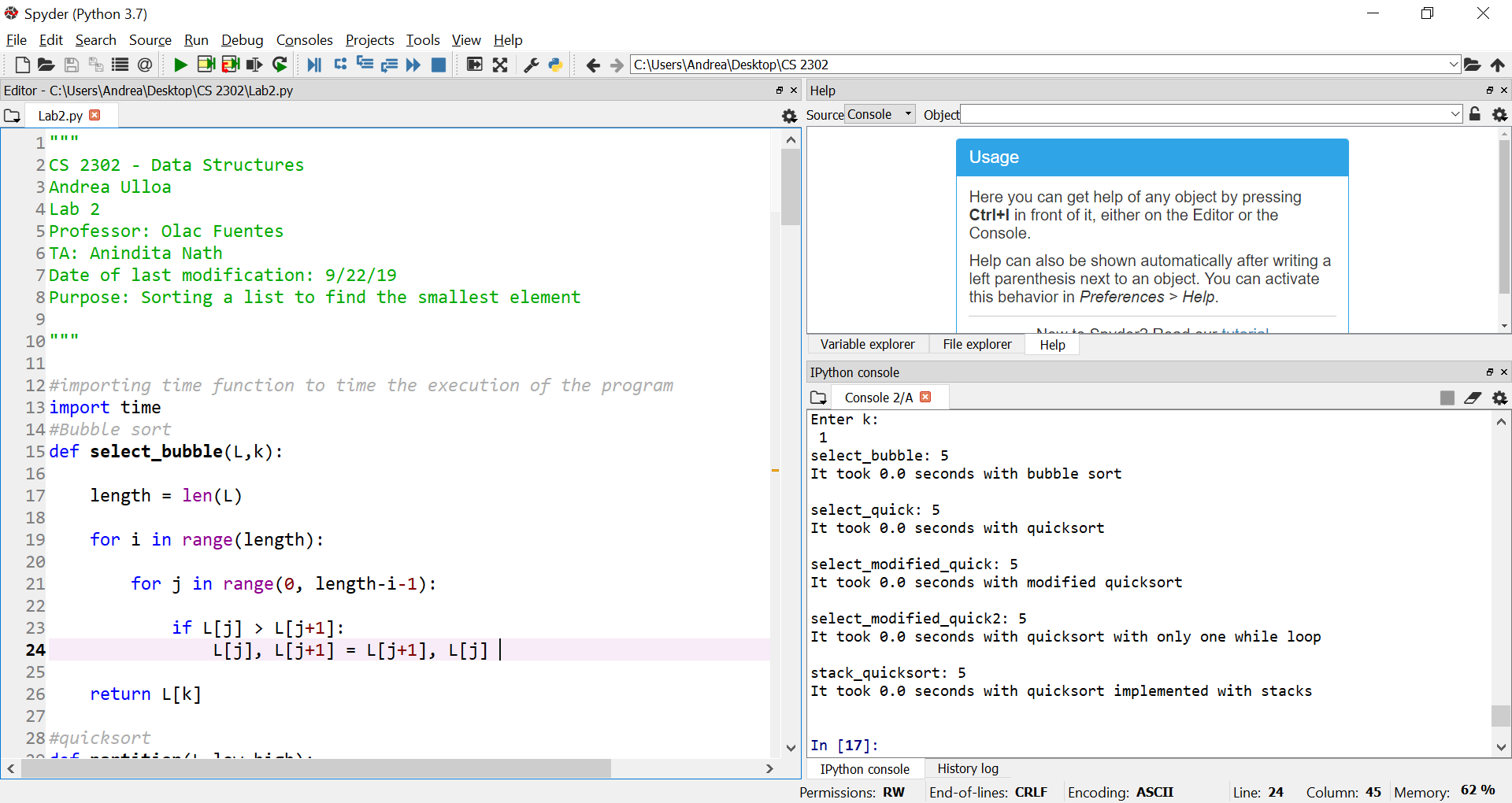
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sorting Algorithm** | **Test Case 1** | **Test Case 2** | **Test Case 3** | **Test Case 4** | **Test Case 5** |
| **select\_bubble()** | 0.0012099 | 0.0 | 0.0 | 0.0002317 | 0.0010008 |
| **select\_quick()** | 0.0 | 0.0003874 | 0.0 | 0.0 | 0.0 |
| **select\_modified\_quick()** | 0.0 | 0.0010244 | 0.0001804 | 0.0 | 2.3841857 |
| **select\_modified\_quick2()** | 0.0 | 0.0 | 0.0 | 0.0010769 | 0.0 |
| **stack\_quicksort()** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0002195 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

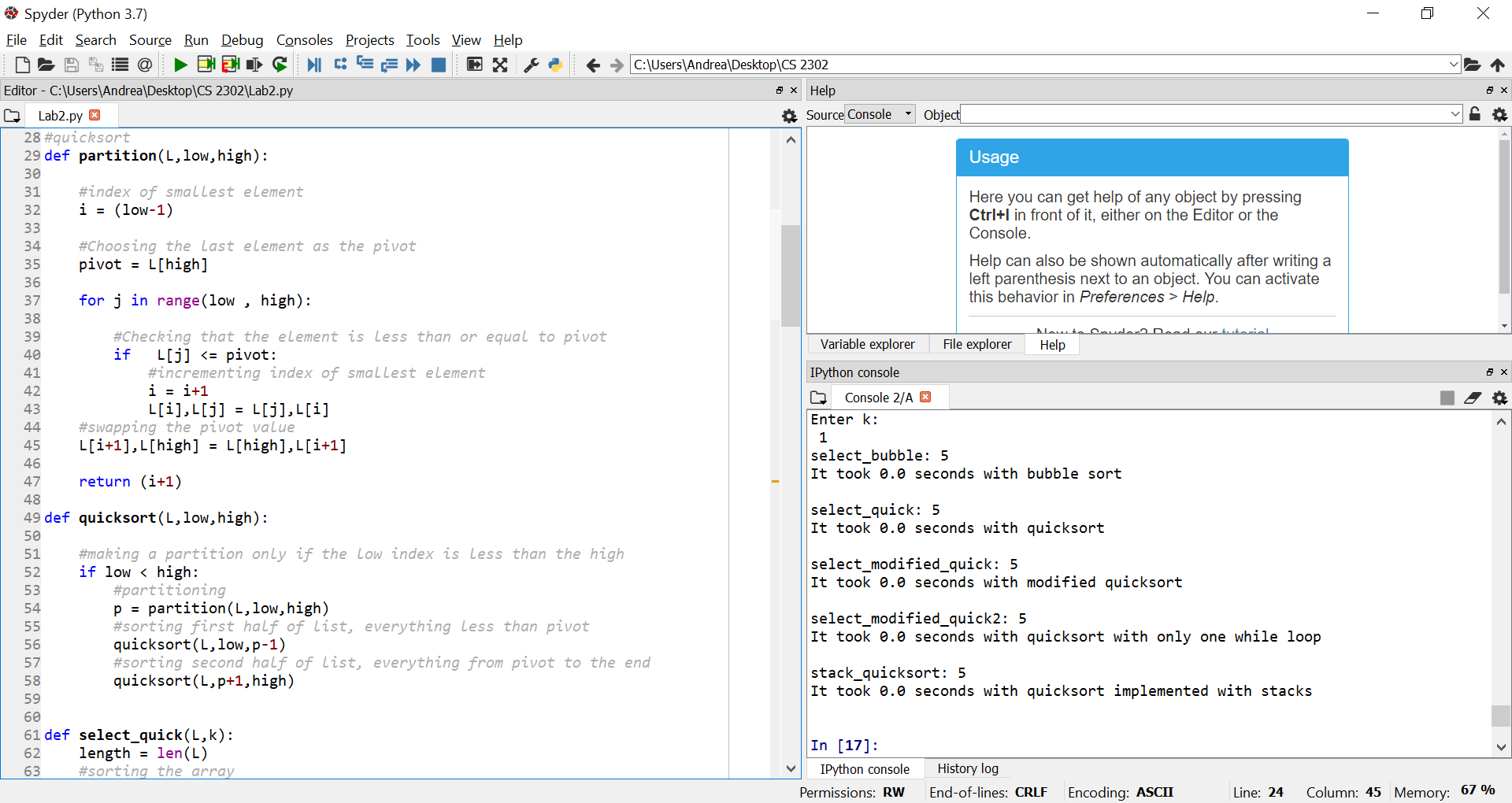
Overall, the quicksort implementation with stacks was the algorithm that performed the operations the quickest. The next efficient algorithm was the regular quicksort which overall ran for 0 seconds except for one of the test cases. This occurred since the function simply stacks every element of the list and orders it without having any unnecessary checks that take more time. The regular quicksort contains a Big-O running time of O(logn) which is relatively faster compared to the other running times. This is because the list is split in two and there are only two recursive calls that make the sorting more efficient. Bubble sort and the modification of quicksort were the slower algorithms due to the Big-O running times of O(n).

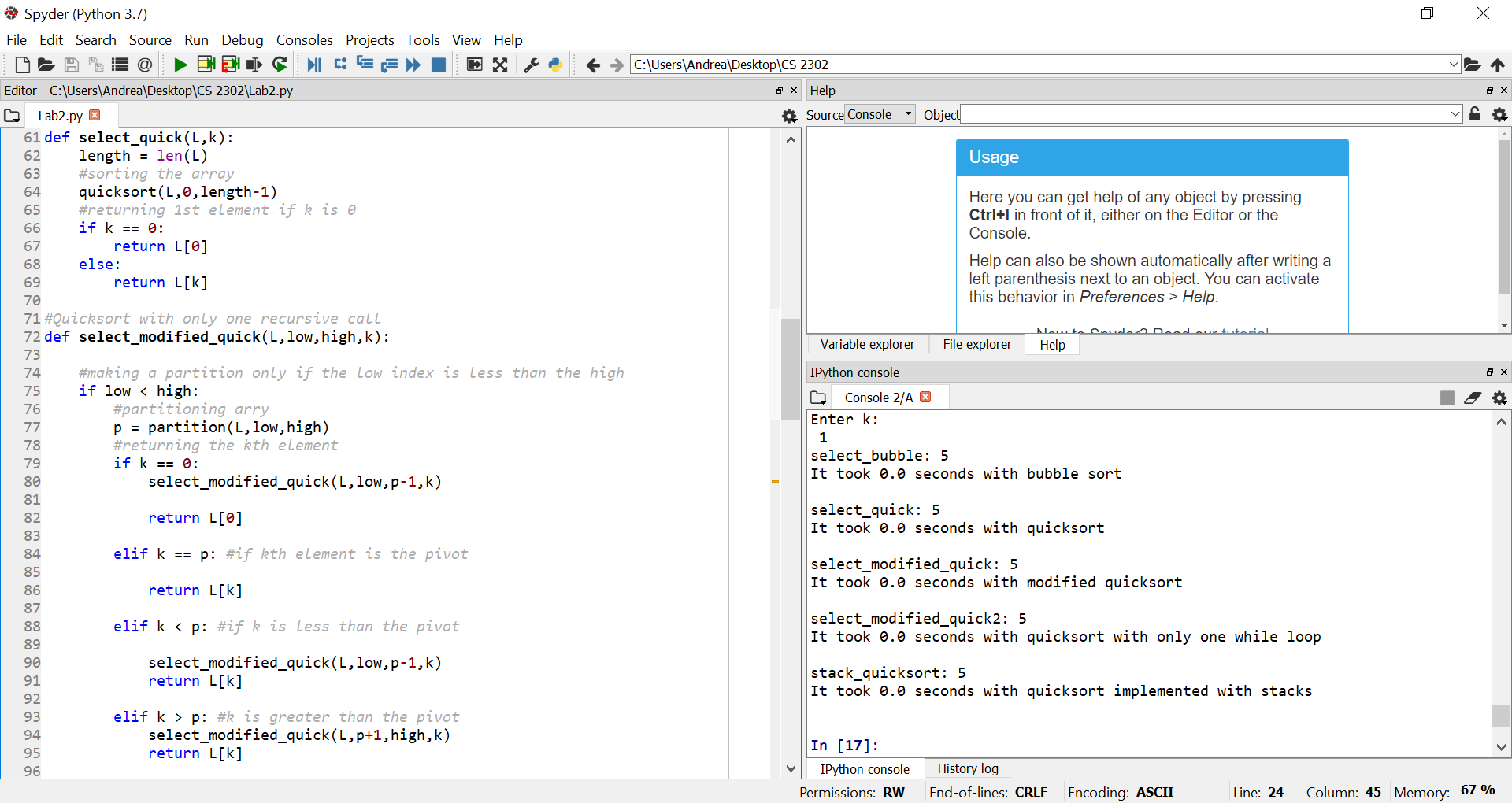
**Conclusion**

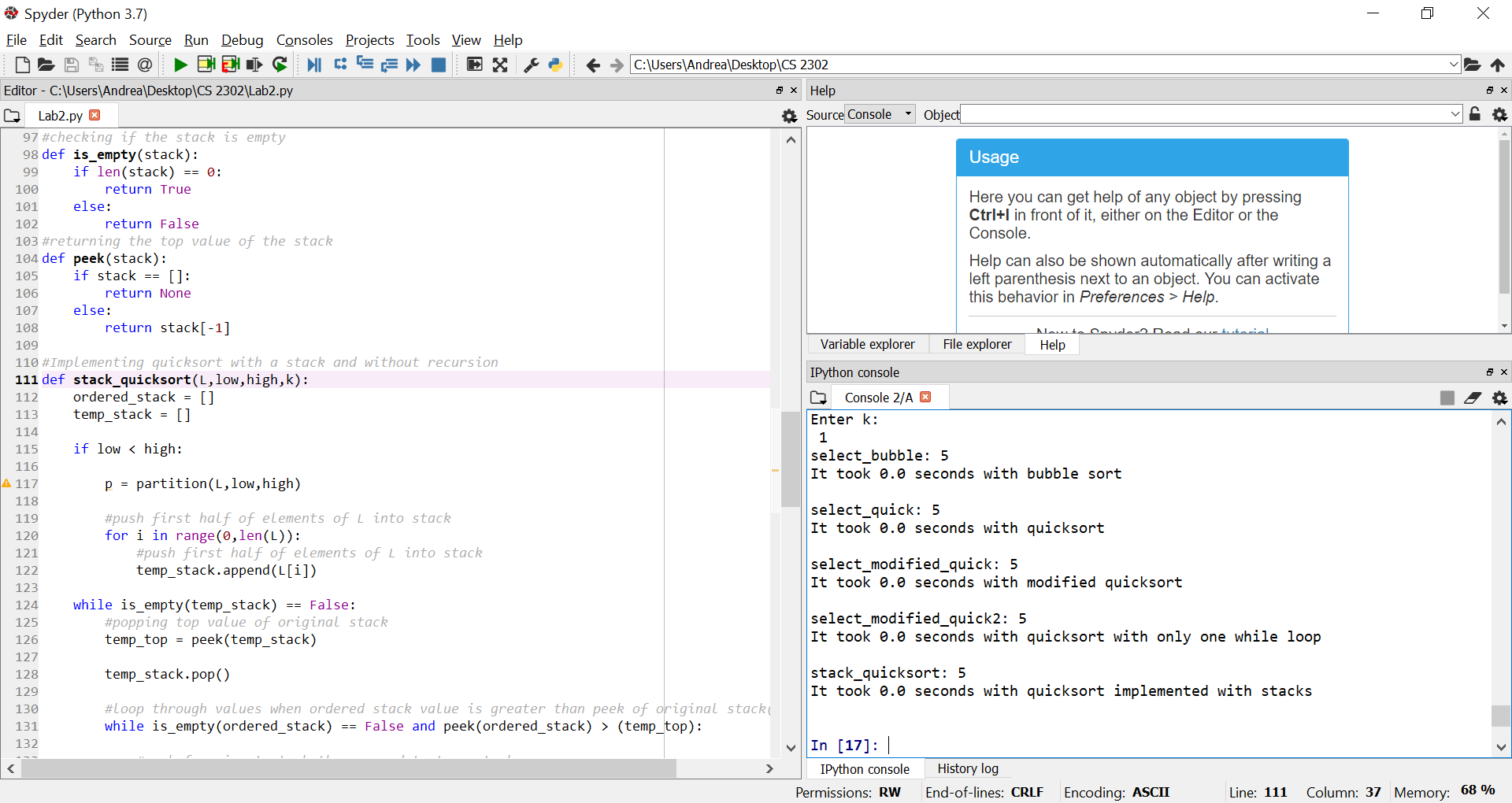
In this lab I learned about different ways of implementing sorting algorithms. Bubble sort and quick sort were the main algorithms in this lab, and I got a better understanding of both of them. I learned how to implement these sorting algorithms by using recursion, and also by creating iterative solutions. I also learned how to implement stacks with quicksort and got a better understanding of quicksort as a whole. Comparing Big-O running times and pinpointing the quickest algorithms for every experiment also helped me learn more about how certain operations can really affect the efficiency of the execution of certain algorithms.

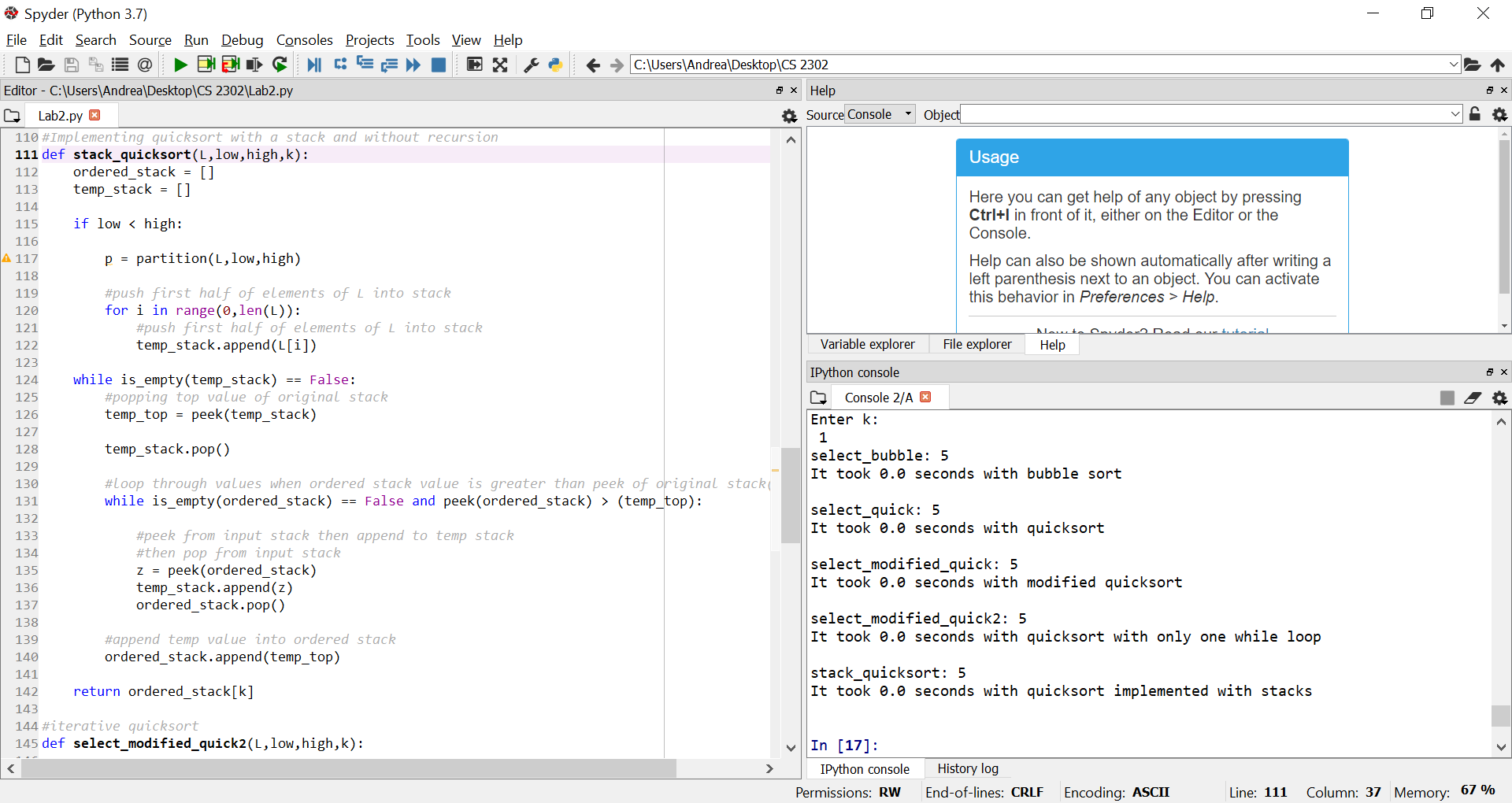
**Appendix**

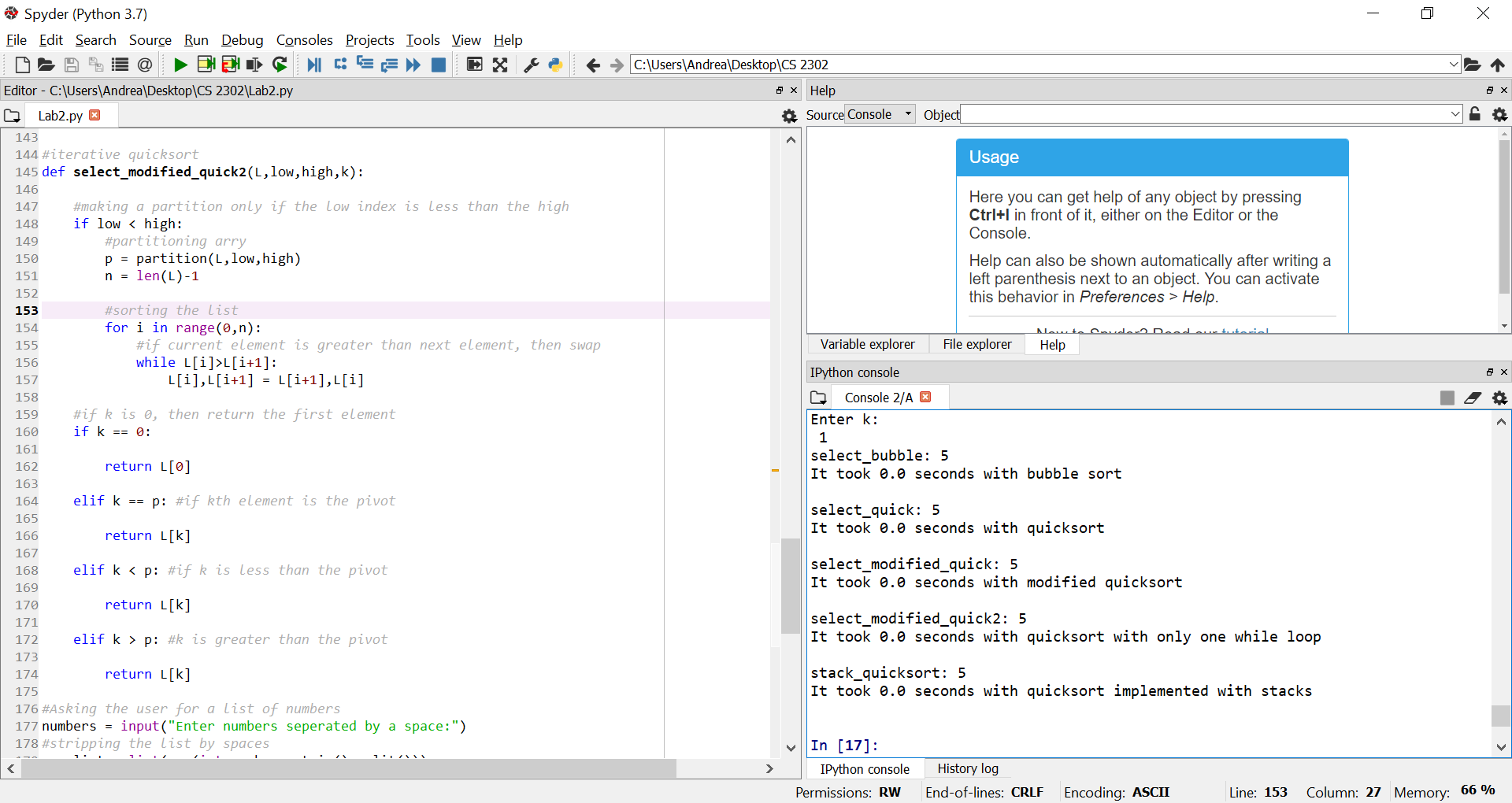


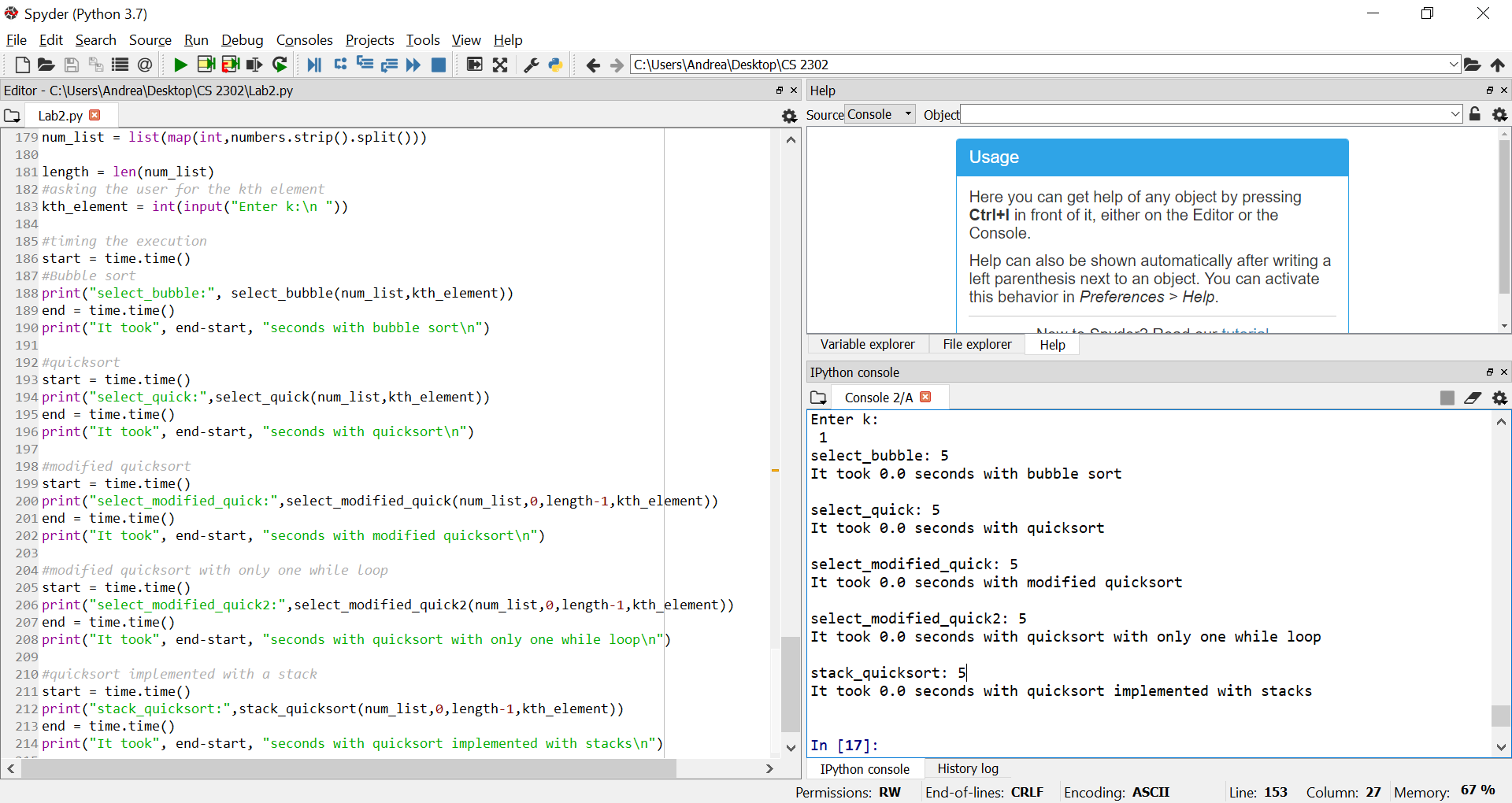












I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class